Automatic Synthesis of Schedulers in Timed Systems [1]

Padmanabhan Krishnan

Department of Computer Science
University of Canterbury, PBag 4800
Christchurch, New Zealand
paddy@cosc.canterbury.ac.nz

Abstract

In this article we present a synthesis technique for generating schedulers for real-time systems. The aim of the scheduler is to ensure (via restricting the general behaviour) that the real-time system satisfies the specification. The real-time system and the specification are described as Alur-Dill timed automata while the synthesised scheduler is a type of timed trajectory automaton. We also note a simple constraint that the specification has to satisfy for this technique to be useful.

1 Introduction

There is no doubt that scheduling plays a central role in the development of real-time systems. There is a large body of knowledge in this area [13,6]. The principal focus of this work has been to study the behaviour of general algorithms (such as earliest deadline first, rate monotonic scheduling etc.). This then identifies a class of real-time applications for which a particular algorithm is ideal (e.g., results in optimal behaviour). A particular approach to scheduling is the static priority assignment method [3]. Here each task is assigned a priority which determines when enabled tasks are executed. For this approach to be successful, two classes of problems need addressing. The first is validation, i.e., checking the priorities assigned are satisfactory and no timing constraint is violated. The second is priority synthesis which actually finds an assignment for a given set of tasks and timing constraints.

If the synthesis algorithm ensures validity, one needs to perform only the synthesis. The time to perform validation is thereby saved. That is, the proof of correctness for the synthesis algorithm needs to be carried out only once.

[1] Extended Abstract: Only Proof Outlines are Presented

After that it can be applied, without proof, to the appropriate class of systems. This is in contrast to the approach [16] of verifying a particular algorithm.

However, the usual synthesis algorithms are conservative in that they may not be able to find a priority assignment even though one may exist. This is mainly because priorities are not sufficiently expressive. One of the ongoing projects is to find algorithms which are less conservative and yet safe.

These scheduling algorithms are dependent on the chosen model of real-time systems. The usual model is to classify the various tasks as either periodic or sporadic, and asynchronous or synchronous tasks. Also associated with the tasks are the real-time features such as deadline, jitter etc. [5]. In such task models some of the key questions that need to be addressed include (a) will a given task meet its deadline and (b) can a high priority task be blocked by a low priority task.

The importance of automatic synthesis is enhanced with the increase in complexity of system requirements. [12] describe an object-oriented technique based on the automata model of statecharts. They do not describe a scheduler *per se*. Rather a priority based scheduler is assumed. They assign dynamic priorities to multi-threaded executions based on the real-time constraints. To automate this process they consider a limited type of constraints such as end-to-end deadlines and activation periods. [12] argues for the advantages of the synthesis method over the user specified technique. That is, if the synthesis technique is not used, the programmer has to encode the scheduler which should ensure that various requirements are met. In a formal specification setting, the specifier has to specify the behaviour of the scheduler in an abstract fashion [8]. The composite system then has to be verified.

In this article we take a different approach to the problem of scheduling in real-time systems. The main purpose of a scheduler is to ensure that the overall system (consisting of many tasks) meets various requirements (such as tasks do not miss deadlines). This is achieved by the scheduler controlling the execution of the potentially competing tasks with conflicting requirements. This is not very different from the notion of a controller that keeps the behaviour of a plant within a given specification [11] in discrete event systems (DES).

[11] show that if the plant is modelled as a finite automaton and safety is expressed as reachability, one can automatically synthesise "a most general controller". That is, the most general controller permits all possible legal behaviours. This synthesis algorithm is based on the classical two person game approach using a fixed point computation for reachability. We adapt the idea of synthesising a controller from plant and specification descriptions [11] to obtain schedulers.

The main difference is that in our system schedulers are reasonably general and do not necessarily have detailed knowledge of the plants. In our model, schedulers know only about preemption points and the number of tasks in the system. They do not have information on the behaviours of the tasks. The synthesis process uses the behaviour of the tasks to generate the sched-

uler. The simple structure permits the scheduler to be implemented on PLCs where the preemption points are inputs and the identifiers of the tasks to be enabled form the outputs. Due to the simpler structure, the technique will be applicable only to certain types of systems.

There exist two main extensions of the controller based ideas to timed systems. [4] treats time as a discrete entity. In this case, the fundamental techniques are no different from the classical approach. [9] presents a synthesis technique where time is a continuous entity. However the actions are still discrete. This model is based on the timed automata model [1]. The synthesis process is based on clock regions (or time equivalences that cannot be distinguished by the clock constraints). Hence it is exponential (using the largest clock values specified and the number of clocks) in the size of the timed automata. As mentioned earlier we require schedulers to have a simpler structure than controllers so that they can be easily implemented. In terms of automata, the size of the alphabet of a scheduler automaton is smaller than the size of the alphabet of a controller automaton. In both these works the aim is to synthesise a real-time controller which can use detailed knowledge of the plant.

The notion of using trajectories [10] to define the parallel composition of words leads to a rich, robust and general theory. A regular trajectory can be viewed as a finite automaton over a simple alphabet. We extend the ideas underlying trajectories to synthesise schedulers for real-time behaviour.

In summary, we use the timed automata model to describe a real-time system, use ideas from DES to describe preemption points, and extend trajectories to describe scheduling. The key result is that under certain conditions one can automatically generate a scheduler. In the next section we quickly review the relevant background material on trajectories, discrete event systems and timed automata. In Section 3 we present our approach and in Section 4 we present a few results.

2 Background

We first review the concepts underlying regular trajectories. This is followed by a more detailed summary of concepts underlying real-time automata. We finally present the key issues for the synthesis of controllers in DES.

2.1 Trajectories

A trajectory represents a controlled shuffle of two words or languages. This controlled shuffle can be viewed as a walk in a two dimensional plane. It can also be viewed as the execution of a scheduler over a two task system. To describe this syntactically the alphabet $V = \{r, u\}$ is used. The symbol r denotes moving right and the symbol u indicates moving up. When the shuffle is written in a linear fashion, r denotes obtaining a symbol from the left and

u denotes obtaining a symbol from the right. On a two dimensional plane the string on the left is along the x axis and the string on the right is along the y axis. A *trajectory* is an element of V^*. The *shuffle* of two words using a trajectory (indicated by $\mathbin{⧢}$ instead of $\sqcup\!\sqcup$) is defined as follows:

- $aw \mathbin{⧢}_{rt} w' = a(w \mathbin{⧢}_t w')$
- $w \mathbin{⧢}_{ut} aw' = a(w \mathbin{⧢}_t w')$
- $\epsilon \mathbin{⧢}_{\epsilon} \epsilon = \epsilon$
- All other cases are undefined.

For example, $a_1a_2 \mathbin{⧢}_{ruuur} b_1b_2b_3$ will yield $a_1b_1b_2b_3a_2$ while $a_1 \mathbin{⧢}_{uurr} b_1b_2$ is not defined as there is no symbol after a_1 that can be selected by the second r.

One can extend the definition to sets of trajectories, where $\alpha \mathbin{⧢}_T \beta = \bigcup_{t \in T} \alpha \mathbin{⧢}_t \beta$. This can be further extended to languages, where $L_1 \mathbin{⧢}_T L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} \alpha \mathbin{⧢}_T \beta$.

If $T = \{r, u\}^*$, then the set of trajectories corresponds to the usual definition of a shuffle. This is because one can select a symbol from either the first component or the second component without any restriction. The trajectory $T = r^*u^*$ defines the concatenation operator as all the symbols from the first string are chosen before any symbol from the second string. All the symbols from the second string are also chosen.

Numerous results concerning trajectories are presented in [10]. The following result is of interest to us. It shows that regular languages are closed under regular trajectories.

Proposition 2.1 ([10]) If L_1 and L_2 are regular languages, $L_1 \mathbin{⧢}_T L_2$ is regular iff T is regular.

2.2 Timed Automata

We now present a quick overview of timed systems. A timed automaton is a finite automaton equipped with a set of timers (or clocks) C. Assume a set of timing constraints ($\Phi(C)$) expressed over C as follows:

$$\beta ::= x \leq n_0 \mid n_0 \leq x \mid \neg\beta \mid \beta_1 \wedge \beta_2$$

where x is a timer in C and n_0 a rational constant. Hence a timed automaton is a structure $(Q, \Sigma, C, \longrightarrow, S, F)$, where Q is the finite set of states, Σ the input alphabet, C the set of available clocks or timers, $\longrightarrow$ the transition relation, $S \subseteq Q$ the set of start or initial states, and $F \subseteq Q$ a Büchi acceptance condition.

An element of the transition relation of the form $q \xrightarrow[X]{a,\beta} q'$ indicates that the automaton can move from state q to state q' on the input a provided the current timer values satisfies the timing constraint β. When the move is made

all the timers mentioned in X are reset to 0.

In order to formally specify the behaviour of such an automaton, a notion of *extended state* is useful. Given a set of clocks C, a time valuation ν is a map from C to $\mathbb{R}^{\geq 0}$. Let V represent the set of all timer valuations. We define $Q \times V$ to represent the set of extended states. That is, an extended state represents the state of the automaton as well as the values held in the various timers.

The acceptance behaviour of timed automata is characterised by timed words. A timed word is a pair (α, τ) where α is an infinite sequence over Σ and τ an increasing non-Zeno sequence over the positive reals ($\mathbb{R}^+$). A *run* of an automaton over such a word is a map $\rho : \omega \longrightarrow Q \times V$ such that the following conditions hold. We write (q_i, v_i) for $\rho(i)$.

- $\rho(0) = (q_0, v_0)$, where $q_0 \in S$ and $v_0(c) = 0$ for every clock c.
- for every $i \geq 0$ there exists β and X such that
  - $(q_i, \sigma_i, \beta, X, q_{i+1})$ is a valid transition
  - $v_i + \tau(i) - \tau(i-1)$ satisfies β
  - $v_{i+1}(c) = 0$ if $c \in X$, otherwise $v_{i+1}(c) = v_i(c) + \tau(i) - \tau(i-1)$

A run identifies the duration that elapses between two actions. The clocks are updated appropriately and the updated clocks are used to determine if the timing constraint is satisfied. If the transition is taken, the relevant clocks are reset to 0. Such a run is *accepting* if the set of states visited infinitely is not disjoint with F, i.e., the Büchi acceptance condition is satisfied.

A machine is *deterministic* if for every state, every next letter to read, and every possible value of clock, the next state is unique. In this case an automaton can have only one run over a given word.

To illustrate some of these definitions, we discuss a simple example. Consider the automaton shown in Figure 1. The automaton accepts an infinite sequence of abc. There are no timing constraints on a. However, whenever the a occurs the timers x and y are reset. The action b can happen only after at least 5 units of time after the occurrence of a has elapsed. However, c must occur within 7 units of time after a. Formally it accepts the language $((abc)^\omega, \tau)$, where $\tau_{3i+1} - \tau_{3i} > 5$ and $\tau_{3i+2} - \tau_{3i} < 7$.

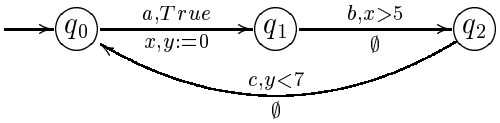


Fig. 1. Sample automaton

For the sake of simplicity we will consider only finite behaviour and treat F as the set of final states. This also means that the condition of time being non-Zeno is not relevant. Furthermore, we will assume that the systems are deterministic. This simplifies the requirements on the specifications, as well

as the various proofs. It is possible to extend these basic ideas of scheduler synthesis for infinite behaviours (ω languages) using the concepts developed in [14,15].

2.3 Discrete Event Systems

Given a description of a plant (P) (or the environment) and a desired specification (S), the purpose of a controller (C) is to ensure that the operation of the plant is within the desired specification. That is, the behaviour of (P $\|$ C) should be contained in S. Modelling P, S and C as finite state automata results in a powerful and useful theory. There are certain restrictions on C for it to be called a controller for P. Every symbol in the alphabet of the automaton P can be classified as either controllable or uncontrollable. A controllable action can either be disabled or enabled by the controller. An uncontrollable action cannot be disabled by the controller. Hence in any global state of (P $\|$ C) (reached after exhibiting w), if P can exhibit an uncontrollable action (a), C cannot prevent it. Therefore, if the specification permits w, it has to permit wa.

This can be formally stated as follows. Given an alphabet Σ, let Σ_u denote the uncontrollable subset of Σ and Σ_c denote the controllable subset of Σ. A prefix closed language L is a *controllable language* with respect to a prefix closed language G iff for every $\alpha \in \Sigma^*$ belonging to L and a belonging to Σ_u, if αa belongs to G, αa belongs to L. Here G represents the trace behaviour of the plant P and L the trace behaviour of the controller C. In any given state of the computation (denoted by α), if the plant can perform an uncontrollable action (a) the controller must allow it to occur. The above definition can be extended to non-prefix closed languages. A language L is a controllable language with respect to a language G iff the prefix closure of L is controllable with respect to the prefix closure of G.

If the specification is mainly concerned with safety properties (expressed as reachability), one can automatically synthesise 'a most general controller'. That is, the most general controller permits all possible legal behaviours. This synthesis algorithm is based on the classical von-Neumann two person game approach using a fixed point computation for reachability. The reader is referred to [11] for details.

3 Real-Time Schedulers

Rather than describe schedulers (*á la* trajectories) as timed regular expressions [2], we describe the set of trajectories as an automaton. This is because it is easier (and more intuitive) to describe real-time behaviour as an automaton rather than as a timed regular expression.

The key question we address in this paper is, given a collection of timed automata $\mathcal{A}$ and a specification $\mathcal{S}$ (which may be required to satisfy certain

restrictions), can one automatically synthesise an appropriate scheduler $\mathcal{C}_{\mathcal{A},\mathcal{S}}$ such that the behaviours generated by the scheduled set of tasks is contained in the behaviours of $\mathcal{S}$. Before we address the synthesis problem, a few definitions and generalisations are necessary.

Firstly we assume that the alphabet (Σ) of the system (viewed as an automaton) is partitioned into schedulable (Σ^s) and unschedulable (Σ^u) sets. We also associate an index number (between 1 and n) to each schedulable action. This further partitions Σ^s into Σ^s_i for each index value i. In essence we are assuming that the system consists of n automata (tasks) over the specified alphabets.

The above structure arises naturally in many systems. Assume we are given a set of tasks that are independent (i.e., their alphabets are all disjoint). We can represent a collection of n independent tasks as $(\mathcal{A}_1, \mathcal{A}_2, \cdots \mathcal{A}_n)$, where each $\mathcal{A}_i$ is a real-time automaton. We will also assume that the input alphabet of $\mathcal{A}_i$ is Σ_i. We will assume that each Σ_i is partitioned into Σ^s_i and Σ^u_i, indicating the schedulable and unschedulable actions respectively for the given task. We can then use Σ to be the union of all Σ_i, Σ^s to be the union of all Σ^s_i, and Σ^u to be the union of all Σ^u_i. The global system, or the joint behaviour of the timed tasks, is described by the product system of the individual tasks. Denote this by $\mathcal{A}$ which is the global automaton to be scheduled given the information concerning the alphabets.

One could also construct one large system and identify sub-tasks within, thereby obtaining the desired structure on the alphabet. It is easy to see that one can describe a real-time system using one of the standard task models and then automatically convert it to have the necessary structure.

A *scheduler* for such a system will be a timed automaton over the alphabet $\{1, 2, \cdots, n\}$. This is a generalisation of a trajectory over a two task system (where r is represented by 1 and u by 2). We will assume that the set of states and clocks used by the two automata are disjoint. The joint behaviour of a system and a scheduler is defined below.

Definition 3.1 Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma_\mathcal{A}, C_\mathcal{A}, \longrightarrow_\mathcal{A}, S_\mathcal{A}, F_\mathcal{A})$ be a task system and $\mathcal{C} = (Q_\mathcal{C}, \Sigma_\mathcal{C}, C_\mathcal{C}, \longrightarrow_\mathcal{C}, S_\mathcal{C}, F_\mathcal{C})$ be a scheduler. Define the *scheduled system* (written as $\mathcal{A} \between \mathcal{C} = (Q, \Sigma, C, \longrightarrow, S, F)$) as follows :

- $Q = Q_\mathcal{A} \times Q_\mathcal{C}$, $S = S_\mathcal{A} \times S_\mathcal{C}$ and $F = F_\mathcal{A} \times F_\mathcal{C}$.
- $C = C_\mathcal{A} \cup C_\mathcal{C}$
- $\Sigma = \Sigma_\mathcal{A}$
- The transition relation $\longrightarrow$ is as follows:
  - $(q_1, q_2) \xrightarrow[X]{a,\delta} (q_1', q_2')$ iff $q_1 \xrightarrow[X_1]{a,\delta_1}_\mathcal{A} q_1'$ and $q_2 \xrightarrow[X_2]{i,\delta_2}_\mathcal{C} q_2'$ provided $a \in \Sigma_i{}^s$, where $X = X_1 \cup X_2$ and $\delta = \delta_1 \wedge \delta_2$.
  - $(q_1, q_2) \xrightarrow[X_1]{a,\delta_1} (q_1', q_2)$ iff $q_1 \xrightarrow[X_1]{a,\delta_1}_\mathcal{A} q_1'$ and $a \in \Sigma^u$.

The first rule describing the transition system captures the requirement

of permission from the scheduler for a schedulable action to be exhibited, while the second rule specifies that unschedulable actions can progress asynchronously without the permission of the scheduler. Note that the scheduler cannot make perform a transition without the system performing a corresponding transition. The rest of the components follow the standard definition.

Note that we are still assuming a single notion of time. Hence there is no guarantee that the joint scheduled system has any behaviour at all. For example, the scheduler could permit the first component to make a move. After that it might give the second component a chance, by which time the second component may not be able to move at all.

An immediate consequence of our definition is that timed languages are closed under scheduling. Furthermore, it is easy to prove that a scheduler only restricts the behaviour of the task system.

Proposition 3.2 *$\mathcal{A} \Join \mathcal{C}$ accepts a time regular language.*

Proposition 3.3 *$\mathcal{L}(\mathcal{A} \Join \mathcal{C}) \subseteq \mathcal{L}(\mathcal{A})$.*

3.1 Synthesis

The primary aim of the above formalism is to automatically synthesise schedulers. The general synthesis [9] of controllers (via game playing and intersection) in the context of timed systems cannot be completely automatic in practice. The principal problem is that timed reachability analysis is expensive. Such an analysis requires the 'clock region' construction, which is exponential in the number of timers (or clocks) used, and the maximum constant that occurs in the timing constraints. Here we adopt the automata theoretic ideas where the resulting scheduler is not based on regions and hence does not have a large state space. This is because of the limited control given to a scheduler. As the alphabet of $\mathcal{C}$ is $\{1, 2, \cdots n\}$, the usual intersection construction is not directly valid. However, we discuss the technique to adapt it to obtain a scheduler in certain circumstances.

We first discuss the synthesis process, present the requirements on the specification for this process to be correct, and study the various properties it satisfies. Let the collection of tasks be represented by $\mathcal{A}$ and the specification be represented by the automaton $\mathcal{S}$. The first step in the synthesis process is to construct the automaton corresponding to $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{S})$. This construction will ensure that the behaviour of this composite automaton is within the specified behaviour. However, this process does not yield a scheduler (an automaton over the simpler alphabet). This still needs to be constructed and is achieved by a simple relabelling. That is, we can pretend that the intersection automaton is the scheduler, with the labels on the transition structure suitably altered.

The *relabelling* (with two cases to consider) on the transition is performed as follows.

- Each transition of the form $(q_1, q_2) \xrightarrow[X]{a,\delta} (q_1', q_2')$ with $a \in \Sigma_i^s$ is altered to a transition of the form $(q_1, q_2) \xrightarrow[X_s]{i,\delta_s} (q_1', q_2')$ where, X_s is obtained from X by deleting all clocks that belong to the system of tasks and δ_s is obtained by deleting all clock constraints involving clocks from the system of tasks. The deletion of clocks is acceptable as the behaviour we are interested in involves the joint behaviour of the scheduler with the task system.

  This relabelling can be viewed in another fashion. If $q_1 \xrightarrow[X_1]{a,\delta_1} q_1'$ is a possible transition in the task system and $q_2 \xrightarrow[X_2]{a,\delta_2} q_2'$ is a transition in the specification, the scheduler will have the transition $(q_1, q_2) \xrightarrow[X_2]{i,\delta_2} (q_1, q_2')$. Technically, there is no need to construct the intersection automaton. The presence of the intersection automaton helps to simplify the underlying concepts.

- Each transition of the form $(q_1, q_2) \xrightarrow[X]{a,\delta} (q_1', q_2')$ is replaced by a transition of the form $(q_1, q_2) \xrightarrow[\emptyset]{\epsilon,true} (q_1', q_2')$, provided $a \in \Sigma_i^u$.

The first relabelling follows the idea that a scheduler cannot control individual actions (it can only enable processes at preemption points). The second relabelling follows from the fact that unschedulable actions cannot be enabled or disabled. Following the construction given in [7], the relabelled automaton can be converted to an ordinary timed automaton. This is because the ϵ transitions in our relabelled system do not reset any clocks.

This relabelling process is not sufficient in itself to guarantee the correctness of the synthesis process. That is, the essential property that the behaviour of this composite automaton is within the specified behaviour does not hold. One can either alter the synthesis process or impose certain conditions on the specification. We adopt the latter as the conditions we impose are not too stringent and they mirror the notion of controllability. Towards that, we construct an automaton $\mathcal{S}_\epsilon$ from a given specification $\mathcal{S}$. This process is described below.

- Add the transition $q \xrightarrow{\epsilon} q'$ to $\mathcal{S}_\epsilon$ iff there is a transition $q \xrightarrow[X]{a,\delta} q'$ in $\mathcal{S}$ with $a \in \Sigma^u$.
- Add self loops $q \xrightarrow[Y]{a,true} q$ to $\mathcal{S}_\epsilon$ for every $a \in \Sigma^u$ and $Y \subseteq C$.
- For each $b \in \Sigma_i^s$ add a transition $q \xrightarrow[X]{b,\delta} q'$ to $\mathcal{S}_\epsilon$ iff there is a transition $q \xrightarrow[X]{a,\delta} q'$ in $\mathcal{S}$ and $a \in \Sigma_i^s$.

The automaton $\mathcal{S}_\epsilon$ is an extension of the $\mathcal{S}$ in that it treats certain actions as invisible or unschedulable. In other words, an unschedulable action can occur at any time. The timers reset on a particular transition are also not controllable. The first two conditions simulate the effect of an unschedulable action, while the third condition indicates that a scheduler can only perform task level and not action level control. However for schedulable actions, it can

also control the timing behaviour.

Proposition 3.4 $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S}_\epsilon)$

Proof. Every transition on an unschedulable action in $\mathcal{S}$ is simulated in two steps by $\mathcal{S}_\epsilon$. For schedulable actions, the transitions are directly available. □

Based on the definition of $\mathcal{S}_\epsilon$, we can now specify a schedulable language.

Definition 3.5 Let $\mathcal{S}$ denote the specification and L the language accepted by the product of the independent tasks. A specification is said to be *schedulable* with respect to a given system behaviour if the following condition holds.

$$\mathcal{L}(\mathcal{S}_\epsilon) \cap L = \mathcal{L}(\mathcal{S}) \cap L$$

The condition imposed in Definition 3.5 is very similar to the synthesis (especially the relabelling) process. It is also similar to that of controllability in DES.

3.2 Example

We now present two simple examples to illustrate the various issues discussed earlier. Consider the automaton shown in Figure 2. We consider the effect of imposing different alphabet structures on it.

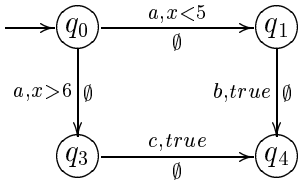
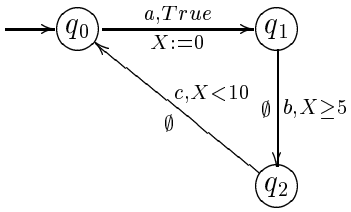


Fig. 2. Simple automaton

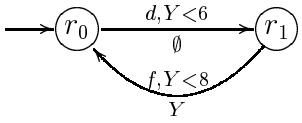
In the first instance let a be the unschedulable action, and b and c belong to Σ_1^s. In that case the specification represented by $p_0 \xrightarrow[\emptyset]{a,y<5} p_1 \xrightarrow[\emptyset]{b,true} p_2$ is not schedulable. This is because the extended automaton $(\mathcal{S}_\epsilon)$ represented by $p_0 \xrightarrow{\epsilon} p_1 \xrightarrow{b,c} p_2$ (a self-loops on p_0 and p_1) in conjunction with the original automaton accepts the word $(a,6)(c,7)$. Note that the a loops are added as a is unschedulable, while the c transition is added as b and c belong to the same task.

If we let Σ_1^s contain only b and Σ_2^s contain only c, the specification shown earlier is schedulable. This is because now the c transition is not added. If we define Σ_1^s to contain only a and make b and c unschedulable, the original specification is schedulable as the timing constraint ensures that the state q_3 is not reached. However, the specification with no timing constraint on a is not schedulable.

Consider the two automata shown in Figures 3(a) and 3(b). Let the specification be that the b should occur before d. The specification as an automaton is shown in Figure 4.

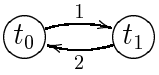


(a) Automaton 1.



(b) Automaton 2.

Fig. 3. Two component automata.

Let a and d be the schedulable actions. It is now easy to see that requirements of schedulability are not met. As the transition on b will be converted to an ϵ move and self loops added, the string $(a, 0.1)(d, 5.5)(b, 6.0)(f, 6.1)(c, 6.2)$ is accepted by $\mathcal{S}_\epsilon$. Hence it violates the definition of schedulability as the specification cannot exhibit this behaviour If b and d are the schedulable actions, the automaton 

acts as a scheduler.

4 Properties

We now present a few properties satisfied by schedulable specifications.

Proposition 4.1 If $\mathcal{L}(\mathcal{S}_1)$ and $\mathcal{L}(\mathcal{S}_2)$ are schedulable with respect to L, $\mathcal{L}(\mathcal{S}_1) \cup \mathcal{L}(\mathcal{S}_2)$ is schedulable with respect to L.

Proof. Follows directly by taking the union of the start states. □

Following [11] it is easy to show that schedulable specifications are *not closed* under intersection. As for DES we let $\overline{X}$ denote the prefix closure of a language X. The following properties capture the notion of schedulability.

Proposition 4.2 If $\mathcal{S}$ is schedulable with respect to L, the following properties hold. As a notational convenience we let S denote $\mathcal{L}(\mathcal{S})$.

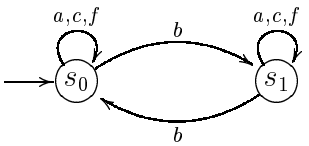


Fig. 4. Specification automaton.

(i) Let $w \in \overline{L} \cap \overline{S}$, $a \in \Sigma^u$ and for some t, $w(a,t) \in \overline{L}$. Then, $w(a,t) \in \overline{S}$.

(ii) Let $w \in \overline{L} \cap \overline{S}$, $a, b \in \Sigma^s$, such that for some t, t' $w(a,t) \in \overline{L}$, $w(a,t) \in \overline{S}$ and $w(b,t') \in \overline{L}$. Then, $w(b,t') \in \overline{S}$.

(iii) Let $w(a,t)(b,t') \in \overline{L} \cap \overline{S}$, where $a \in \Sigma^u_i, b \in \Sigma^s_j, i \neq j$. If $w(b,t'') \in \overline{L}$, then $w(b,t'') \in \overline{S}$.

Proof.

(i) If $w \in \overline{S}$ and $a \in \Sigma^u$ then by the addition of self loops $w(a,t) \in \mathcal{L}(\mathcal{S}_\epsilon)$. As $\mathcal{L}(\mathcal{S}_\epsilon) \cap L = \mathcal{L}(\mathcal{S}) \cap L$, $w(a,t) \in \overline{L}$ requires $w(a,t) \in \overline{S}$.

(ii) The proof for this part is similar to the above proof. The addition of extra schedulable transitions from the same task ensures the result.

(iii) This follows from the fact that the transition of the form $q \xrightarrow[X_a]{a,\delta_a} q' \xrightarrow[X_b]{b,\delta_b} q''$ exists in $\mathcal{S}$, where q is reached while exhibiting w. The construction of $\mathcal{S}_\epsilon$ enables the following transition sequence: $q \xrightarrow{\epsilon} q' \xrightarrow[X_b]{b,\delta_b} q''$ exists in $\mathcal{S}$. Now the result follows directly from the definition of schedulability.

□

Intuitively, the first condition is the usual 'controllability' requirement. That is, any action that does not require the permission of the scheduler cannot be prevented. The second property arises because the scheduler cannot control individual actions. Hence if one schedulable action is permitted to occur, all schedulable actions need to be permitted. The third property is valid because the scheduler only enables schedulable actions and has no information on the other actions. In the above case after the string w is exhibited, the scheduler has to permit a schedulable action from process j. As the scheduler cannot control the unschedulable action from process i, the specification must allow the exhibition of b before a.

It is easy to see that using the definitions of scheduler composition and schedulability, the above synthesis procedure yields a correct scheduler. This is stated formally below. Given the collection of tasks $\mathcal{A}$ and the specification $\mathcal{S}$, we write $\mathcal{C}_{\mathcal{A},\mathcal{S}}$ to denote the scheduler obtained by the above process.

Proposition 4.3 If $\mathcal{S}$ is schedulable with respect to $\mathcal{L}(\mathcal{A})$, then $\mathcal{L}(\mathcal{A} \mathbin{/\!\!/\!\!\backslash} \mathcal{C}_{\mathcal{A},\mathcal{S}}) \subseteq \mathcal{L}(\mathcal{S})$.

Proof. Let us represent a typical state of the joint system $\mathcal{A} \mathbin{/\!\!/\!\!\backslash} \mathcal{C}_{\mathcal{A},\mathcal{S}}$ as $(p_i, (q_i, r_i))$, where p_i and r_i are states of the task system and r_i the state of the specification.

Let the joint system exhibit action a at some t. We have to show that the specification (in an appropriate state) permits this behaviour. There are two cases to consider. The first is if a belongs to Σ^u. In this case the task system has a transition of the form $p_i \xrightarrow[X]{a,\delta} p'_i$ and the scheduler has a transition of the form $(q_i, r_i) \xrightarrow{\epsilon} (q'_i, r'_i)$. This can arise only if the specification had a

transition of the form $r_i \xrightarrow[X']{b,\delta'} r_i'$ for $b \in \Sigma^u$. From the first two extensions on $\mathcal{S}_\epsilon$ it is easy to see that it should also have transitions of the form $r_i \overset{a}{\circlearrowright} \xrightarrow{\epsilon} r_i'$ which can be used to exhibit a at time t. From the definition of schedulability, the specification must be able to exhibit this behaviour.

The second case of a belonging to Σ_i^s can be proven in a similar fashion. For this the third extension is used to show the desired property.

The final result follows directly from the above observations as it is easy to translate a run of the joint system to a run of the specification automaton. □

5 Conclusion and Future Work

In this paper we have described the behaviour of timed schedulers. We have also specified a synthesis technique for real-time systems. For this process to be useful, the specification had to satisfy certain properties.

The schedulability condition as required in Definition 3.5 is sufficient for the process of synthesising a scheduler. However, the condition is too strong and is not a necessary one. The main reason is that, if in a given state the scheduler permits a schedulable action a at time t to occur, it is forced to permit any other schedulable action b at any time t' to occur. That is, the full power of timed behaviour is not exploited.

While a more general definition is possible, it requires the specification to satisfy more complex properties. In practice one needs to verify that the specification satisfies the required properties for the synthesised scheduler to be correct. Hence, a more general synthesis process may spend more time verifying the specification than in generating the scheduler. Further research is necessary to determine the point where these trade-offs (of generality against the time to verify) are acceptable.

Acknowledgements

Thanks to Jane McKenzie for carefully proof reading an earlier draft.

References

[1] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] E. Asarin, P. Caspi, and O. Maler. A Kleene Theorem for Timed Automata. In *IEEE Symposium on Logic in Computer Science (LICS 97)*, pages 160–171. IEEE, 1997.

[3] F. Balarin. Priority assignment for embedded reactive real-time systems. In F. Mueller and Z. Bestavros, editors, *Languages, Compilers and Tools for*

Embedded Systems: LCTES, volume 1474 of LNCS, pages 146–155. Springer-Verlag, June 1998.

[4] B. A. Brandin and W. M. Wonham. Supervisory Control of Timed Discrete-Event Systems. *IEEE Trans. on Automatic Control*, 39(2):329–342, Feb 1994.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, second edition, 1997.

[6] S. C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems: A Brief Survey. In J. A. Stankovic and K. Ramamritham, editors, *Tutorial Hard Real-Time Systems.* IEEE, 1988.

[7] V. Diekert, P. Gastin, and A. Petit. Removing ϵ-Transitions in Timed Automata. In *STACS 97*, volume LNCS-1200, pages 583–594, Lübeck, 1997. Springer Verlag.

[8] J. Jacky. Analyzing a real-time program with z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *Proceedings of ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 136–153. Springer Verlag, 1998.

[9] O. Maler, A. Pnueli, and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In E. Mayr and C. Puech, editors, *Symposium on Theoretical Aspects of Computer Science*, volume 900 in LNCS, pages 229–242. Springer Verlag, 1995.

[10] A. Mateescu, G. Rozenberg, and A. Salomaa. Shuffle on Trajectories: Syntactic Constraints. *Theoretical Computer Science*, 197(1–2), 1998.

[11] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[12] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *Proc. 1997 IEEE Real-Time Systems Symposium, RTSS'97.* IEEE Computer Society Press, December 1997.

[13] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic Publishers, 1998.

[14] J. G. Thistle and W. M. Wonham. Control of Infinite Behaviour of Finite Automata. *SIAM Journal on Control and Optimization*, 32(4):1075–1097, 1994.

[15] J. G. Thistle and W. M. Wonham. Supervision of Infinite Behaviour of Finite Automata. *SIAM Journal on Control and Optimization*, 32(4):1098–1113, 1994.

[16] M. Wilding. A Machine-Checked Proof of the Optimality of a Real-Time Scheduling Policy. In A. Hu and M. Vardi, editors, *Proc. Tenth International Worshop on Computer Aided Verification*, LNCS1427, pages 369–378, Vancouver, Canada, June/July 1998. Springer Verlag.